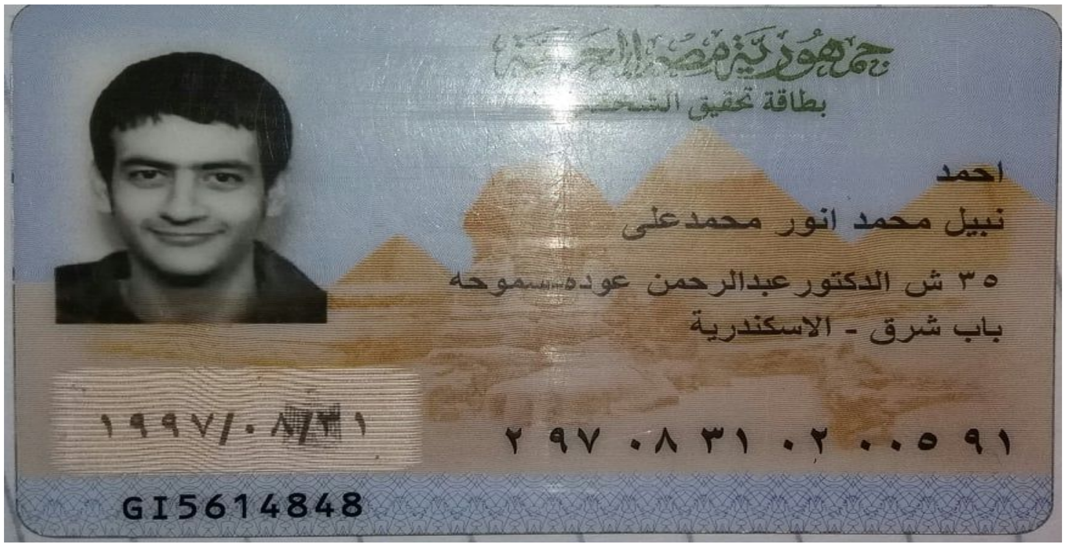
Compiler Project Report

Compilers Final Project

Team Members:

Ahmed Nabil Mohamed Anwar	**(09)**
Abdelrahman Alsayed Ahmed	**(24)**
Mohamed Samy Mohamed	**(40)**
Yehia Elsayed Mohamed Mohamed	**(62)**

IDs



جمهورية مصر العربية

بطاقة تحقيق الشخصية

احمد

نبيل محمد انور محمدعلى

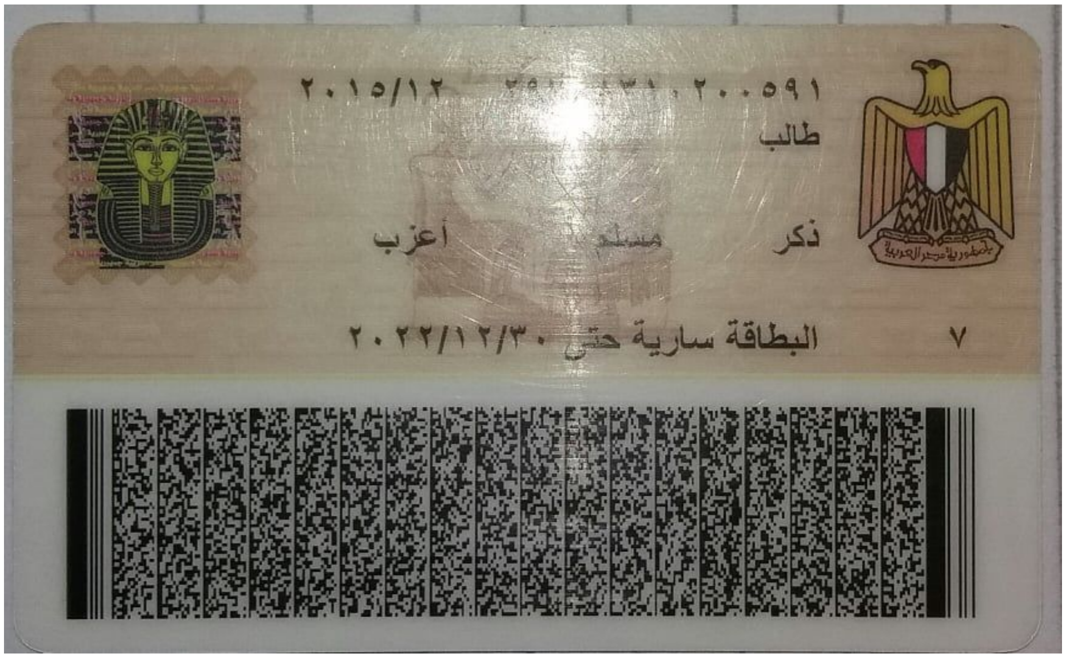
٣٥ ش الدكتورعبدالرحمن عوده-سموحه

باب شرق - الاسكندرية

١٩٩٧/٠٨/٣١

٢ ٩٧ ٠٨ ٣١ ٠٢ ٠٠٥ ٩١

GI5614848

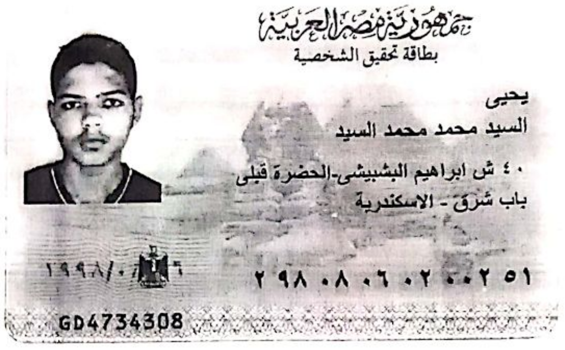


٢٩٧٠٨٣١٠٢٠٠٥٩١ ٢٠١٥/١٢

طالب

ذكر مسلم أعزب

٧ البطاقة سارية حتى ٢٠٢٢/١٢/٣٠



جمهورية مصر العربية

بطاقة تحقيق الشخصية

يحيى

السيد محمد محمد السيد

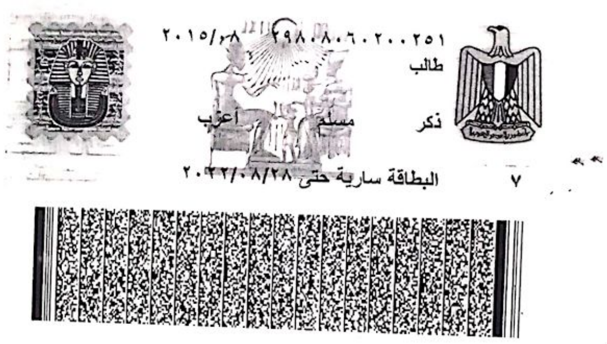
٤٠ ش ابراهيم البشبيشى-الحضرة قبلى

باب شرق - الاسكندرية

١٩٩٨/٠٨/٠٦

٢٩٨٠٨٠٦٠٢٠٠٢٥١

GD4734308

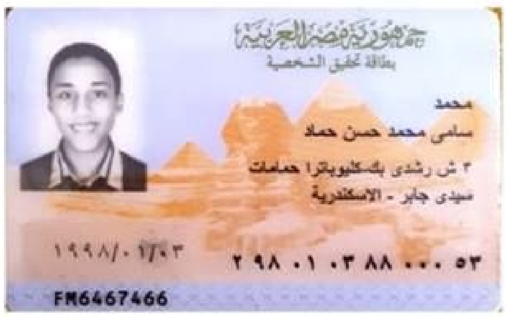


٢٩٨٠٨٠٦٠٢٠٠٢٥١ ٢٠١٥/٠٨

طالب

ذكر مسلم أعزب

٧ البطاقة سارية حتى ٢٠٢٢/٠٨/٢٨



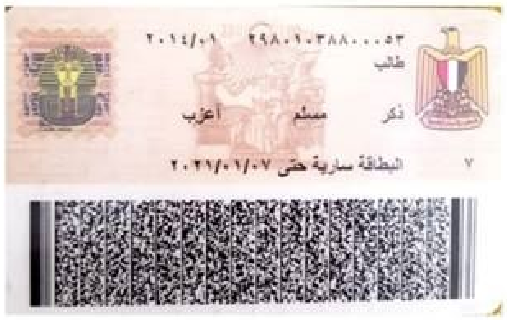
جمهورية مصر العربية
بطاقة تحقيق الشخصية

محمد
سامى محمد حسن حماد
٣ ش رشدى بكسكليوباترا حمامات
سيدى جابر - الاسكندرية

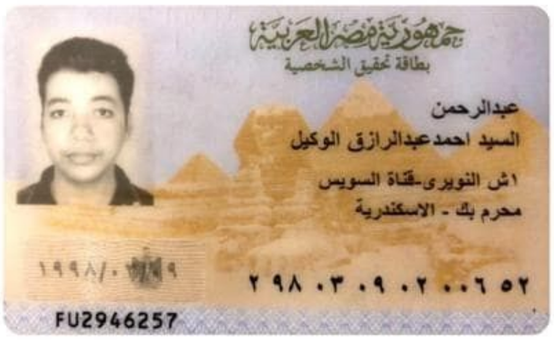
١٩٩٨/٠١/٠٣

٢ ٩٨ ٠١ ٠٣ ٨٨ ٠٠٠ ٥٣

FM6467466



٢٩٨٠١٠٣٨٨٠٠٠٥٣ ٢٠١٤/٠١
طالب
ذكر مسلم أعزب
٧ البطاقة سارية حتى ٢٠٢١/٠١/٠٧



جمهورية مصر العربية
بطاقة تحقيق الشخصية

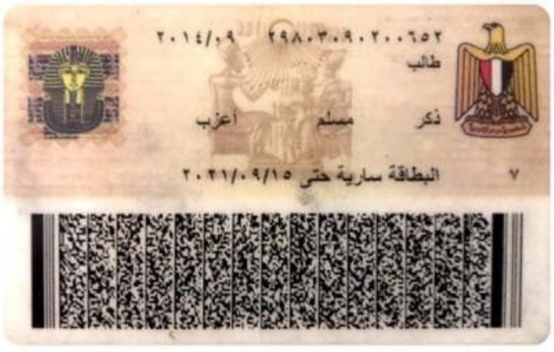
عبدالرحمن
السيد احمدعبدالرازق الوكيل

١ش النويرى-قناة السويس
محرم بك - الاسكندرية

١٩٩٨/٠[illegible]٩

٢٩٨٠٣٠٩٠٢٠٠٦٥٢

FU2946257



٢٠١٤/٠٩ ٢٩٨٠٣٠٩٠٢٠٠٦٥٢
طالب

ذكر مسلم أعزب

٧ البطاقة سارية حتى ٢٠٢١/٠٩/١٥

1. Abstract And Objective

A **compiler** is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an editor.

In this project, it is required to implement the front-end part of a compiler for a chosen subset of the java language and produce the intermediate code of the given input.

This project aims to practice techniques studied during the semester for building automatic lexical analyzer generator tools, automatic parser generator tools, and for constructing semantics rules to generate intermediate code.

2. Description Of The Problem

It is required to develop a suitable Syntax Directed Translation Scheme to convert Java code to Java bytecode, performing necessary lexical, syntax, and static semantic analysis (such as type checking and Expressions Evaluation). This can be divided into 3 parts:

1) **<u>Lexical Analyzer Generator</u>**

   The first part should design and implement a lexical analyzer generator tool. The lexical analyzer generator is required to be tested using the given lexical rules of tokens of a small subset of Java.

2) **<u>Parser Generator</u>**

   The task in this part is to design and implement an LL (1) parser generator tool. The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and uses them to construct a predictive parsing table for the grammar. The parser generator is required to be tested using the given context-free grammar of a small subset of Java. Combine the lexical analyzer generated in part and parser such that the lexical analyzer is to be called by the parser to find the next token.

3) **<u>Java Byte Code Generation</u>**

The task in this part is to write the semantic rules for the given context-free grammar. These semantic rules should output the bytecode. The generated bytecode must follow Standard bytecode instructions defined in Java Virtual Machine Specification.

3. Introduction And Literature Overview

Overview of the three phases :

- **<u>Phase One (Lexical Analyzer) :</u>**

  Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The **lexical analyzer** breaks these syntaxes into a series of tokens, by removing any whitespace or other unused parts in the source code.

  We have built the lexical analyzer generator. It is considered to be the first phase in the compiling process that translates the code into tokens which will help us in the second phase.

- **<u>Phase Two (Parser):</u>**

  A **syntax analyzer** or parser takes the input from a **lexical analyzer** in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

  This phase of the assignment aims to practice techniques for building automatic parser generator tools. The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and uses them to construct a predictive parsing table for the grammar.

- **<u>Phase Three (Java Byte Code Generation):</u>**

  This phase of the assignment aims to practice techniques for constructing semantics rules to generate intermediate code.

  Generated bytecode must follow Standard bytecode instructions defined in Java Virtual Machine Specification.

  We have a lexical file that is used as a substitution of the lexical analyzer in this phase to get the tokens. Parser file that has the semantic rules that can be used to generate the java byte code.

4. Description Of The Used DataStructures

- **Phase One :**

I. Building the NFA

- Unordered_map to save state transitions of every input
- Vector to save the states of the NFA
- Stack used in the evaluation of the regular expression (transfer to postfix expression)

II. Building the DFA

- Vector of strings → final states of NFA, input tags, state tags[Name
- convention of NFA], DFA states.
- Vector of vectors of strings → DFA transitions, Partitioning.
- Map → Final states of DFA with its priority, Grouping of final states that are used in partitioning.
- Unordered_map → Final states of NFA with corresponding priorities.
- Vector of states[Implemented Class] → NFA transitions.

III. Analyzing the Test Code

- The same as NFA & DFA
- Vector of tokens[Implemented Class] → Tokens of the test code.

- **Phase Two :**

I. Building the Rules:

  - Map <String, Rule> → To store each non-terminal and pointer to its corresponding Rule.
  - Vector<String> → To store the productions of each Rule as strings to be used later.

II. Building the Predictive Table:

  - vector<vector<string>> → To store the entries of the predictive table.
  - unorderd_map<string, int> → To store each non-terminal and each terminal name and index in the predictive table.

III. Data Modeling:

  - Classes to store the required information about each rule to improve the running time and applying the OOP design concepts.

- **Phase Three :**

  - **Map <String, String>** → for holding the operations symbols [Int, Float, Arithmetic] and their java byte code to be used in the generation of the code.

  - **Vector<String>** → it will contain the output that will be written (the byte code) and it helps in making actions and the semantic rules.

  - **Map<String, Pair<int, String>>** → this map holds the variables that are declared in the code and also holds their type and address and works in the code as the symbol table also helping a lot in semantics and actions.

5. Explanation of all algorithms and techniques used

- **Phase One :**

I. Building the NFA

- **Thompson's construction**: a method of transforming regular expression to NFA.

- **Postfix expression**: used to sort the operations done in the regular expression.

II. Building the DFA

- **The process of building DFA without minimization** is implemented as described in the lecture by taking the initial state of NFA, get epsilon closure of it, this state will be the initial state of DFA and applying inputs to our new initial states with getting epsilon closure of the result. If the resulting state is defined for the first time, we put it into our transitions table of DFA and repeat previous steps on it and so on.

- **Minimization's technique**: Partitioning method. construct the initial partition pi of the set of states with two groups {final states, non-final states}, apply the partition method on pi, to check if each state is in its right state or need to be moved to another partition using some helper functions. The loop continues until there is no change in the partition . for each group of states choose one state to be the representative for that group. remove the dead states and substitute each state in the transition table with its group representative.

III. Analyzing the Test Code

- Use **maximal munch** to detect the longest valid prefix in the input as a token and then go to the next character after the end of the last token

- Use **panic mode recovery** when no match is found by ignoring the first character and try to match from the next one.

- ## Phase Two

❖ Remove Left Recursion :

Using the algorithm of replacing each non-terminal from the last rules in current rules by its productions. And checking if it has immediate left recursion if so it will remove the left recursion according to the lectures by building the new rule for the new non-terminal and add it to the map of the rules.

❖ Left factoring :

We have sorted vector<string> caring the productions of each rule then splitting each production by the space in a vector<vector<string>> in which each row represent a vector<string> of split production, then we have to check in each section of the productions beginning with the same letter for the first components of each product that is common along biggest no of productions recursively, then adding the new rule for the common factor if found.

- ❖ **Compute first :**

  Compute first of each rule recursively as there is no left recursion by looping throw every production of the rule and check for the first component of the production

  - If it is terminal: add it to the first of the rule
  - If it is nonterminal: solve it recursively and add its first to the first of the current non-terminal.

- ❖ **Compute Follow :**

  Using DFS and backtracking algorithms calculate the following set of each non-terminal by using the first sets in case of needing it. Adding $ sign to the following set of the first non-terminal and continue with the algorithm computing the rest of the following sets as discussed in lectures.

- ❖ **Build the predictive table :**

  We have used the algorithm of iterating through each rule with its name(Non-terminal symbol) and checking for the first terminal symbols of each production to fill its entry in the table with the production rule also looking for the epsilon in each first set and if it is found we also add the production rule in the entry of the rule and its follow terminal symbol. In the end, if there is no epsilon in the first set of each production rule in the rule we use its entries with the following terminal symbols as synch tokens. Any other undefined entries are considered as errors.

- **Stack procedure and error handling :**

  We have used the general algorithm for accepting the inputs of tokens. We have used mainly stack and vectors of tokens. Also, we handled different types of errors(Terminals are not identical - Existence of “Error” in the predicative table) by taking the suitable action that is mentioned in the lecture under the headline(Panic-Mode Error Recovery).

- **Phase Three**

  - **Handling Expressions and declaration :**

    Firstly, we have generated suitable semantics rules that will be associated with our context-free grammar. Also, we add suitable actions with our regular expressions by making use of ‘%union’ [To capture the value of a token if needed]. We have made all the semantics rules that cover the required points in Phase 3:

    - **Primitive types (int, float)** → We accept the declaration, assignment, and take the suitable action that is associated with CFG grammars through two phases. The first is to accept only the declaration and define this variable, check if it exists, and report the error if it exists. The second is to accept the assignment by checking if ‘ID’ has been defined before or not. Also, we accept numbers with -ve signs.

- **Boolean Expressions** → We only support (True, False, RELOP) by taking suitable actions for the true lists and false lists. For ‘True’ the true list for the expression will be the address of ‘True’ terminal, but for ‘False’ we will add the address to the false list. For ‘RELOP’ firstly, we check if we deal with integer numbers or float numbers, then we push the suitable java byte code depending on the operation and the numbers are integer or float.
- **Assignment & Arithmetic Expressions** : first for the assignment if the variable does not exist in symbol table or there is mismatch between value and variable error will be given else it will be saved into the code vector with its java byte code (istore/fstore).For Arithmetic expression we check for the type mismatch to give error if exists else will get the appropriate arithmetic symbol from the maps then add its java byte code to (i/f) and put them in the vector code , term in expression can be values without variable it will also have its byte code.

❖ **BackPatch :**

- Use backPatch to calculate the address of the goto instruction in one pass only
- Every statement, If, while and boolean expression has truelest, falselest, next
- Truelest and falselest have lines numbers of goto statements that need the address of where to go when the condition is true in truelist and false in false list
- Next List contains the lines numbers of the next instruction after statement, if or while

- So when we add goto statement we make the address blank temporarily until it will be filled after that

❖ Handling If :

- When we go to the statement in the if condition then we will fill the the addresses of the goto statement in the truelist lines that need the address to jump when condition is true as this address is what we have now
- When we go to the statement in the else then we will fill the the addresses of the goto statement in the falselist lines that need the address to jump when condition is false as this address is what we have now
- After executing the statement in the true condition or the false condition when needed to go to the next of the if statement. So we combine the next list of the two statement in the next of if
- After returning from if we set the goto statement instruction in the next list of if by the address we have now.

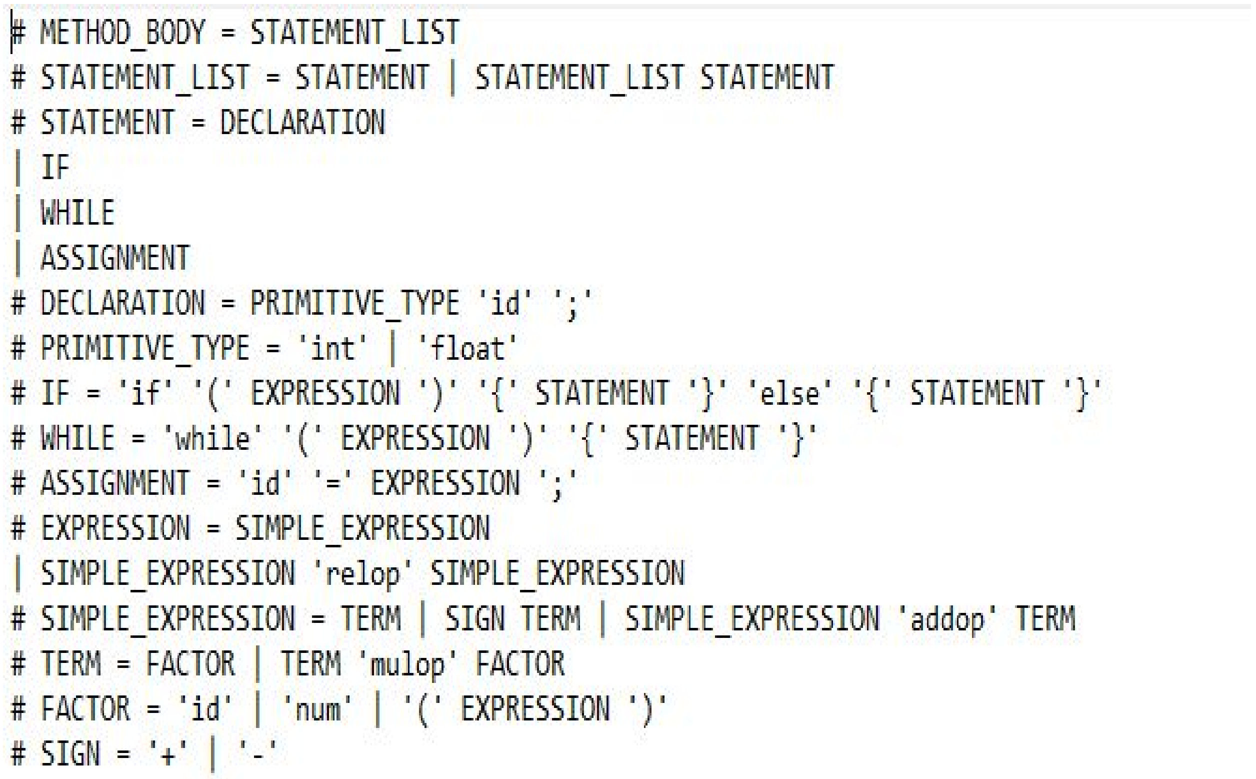
❖ Handling While :

- At first we add mark before translating the boolean expression to return back to at the end of each cycle of the while
- Truelist and falselist of the boolean expression will be like the if statement
- Make the next of while as the false list of the boolean expression and goto the mark to execute the condition again
- After while we add the address of the next list as the address we have after return from the while

6. Sample Runs

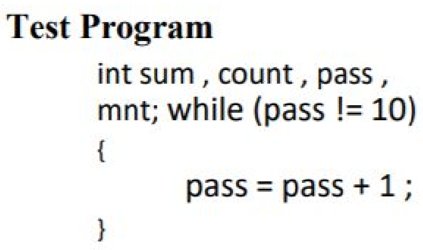
- **Phase one**

⇒ Rules :



```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT = DECLARATION
| IF
| WHILE
| ASSIGNMENT
# DECLARATION = PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE = 'int' | 'float'
# IF = 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
# WHILE = 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
# ASSIGNMENT = 'id' '=' EXPRESSION ';'
# EXPRESSION = SIMPLE_EXPRESSION
| SIMPLE_EXPRESSION 'relop' SIMPLE_EXPRESSION
# SIMPLE_EXPRESSION = TERM | SIGN TERM | SIMPLE_EXPRESSION 'addop' TERM
# TERM = FACTOR | TERM 'mulop' FACTOR
# FACTOR = 'id' | 'num' | '(' EXPRESSION ')'
# SIGN = '+' | '-'
```

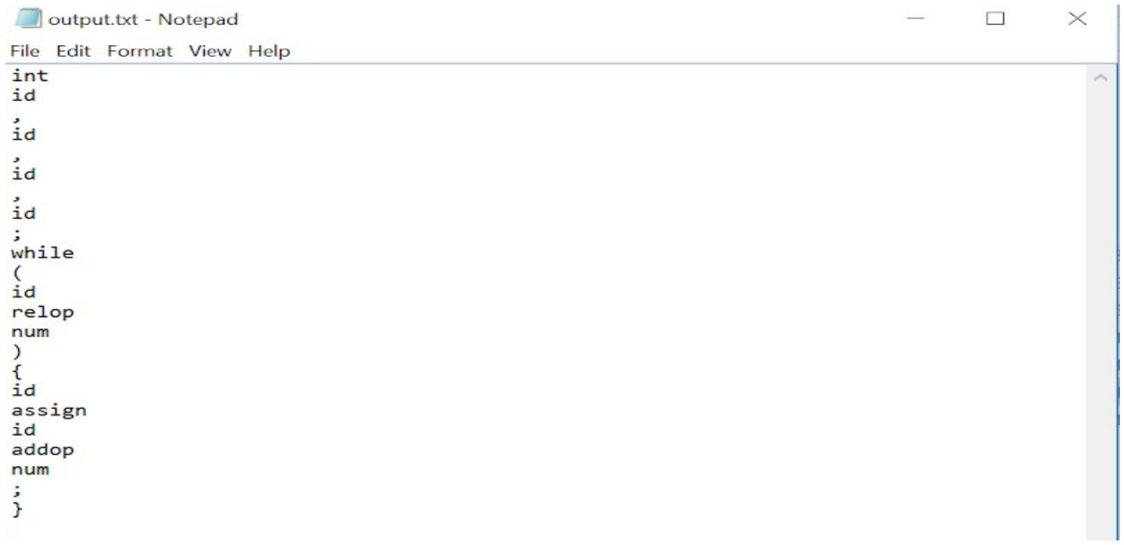
⇒ Input Code :



Test Program

```
int sum , count , pass ,
mnt; while (pass != 10)
{
        pass = pass + 1 ;
}
```

⇒ Resultant of the stream tokens for example program :



```
int
id
,
id
,
id
,
id
;
while
(
id
relop
num
)
{
id
assign
id
addop
num
;
}
```

⇒ Resultant of the stream tokens for another program

https://drive.google.com/open?id=1AixAAEd6wkaM6zUgWDsfxWPEZHpUAEqR

⇒ Test Code of the other program

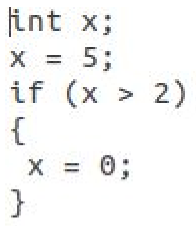
https://drive.google.com/open?id=1tylFGqzhjVyv1KkN88GkA21BiL52u7ft

**** Results are provided with links due to the test code is very long to capture them as screenshots ****

- **Phase Two**

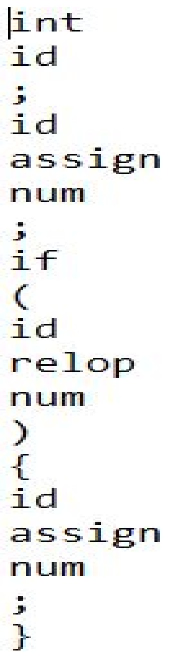
⇒ **Lexical Rules: The same as phase One.**

⇒ **Input test program :**



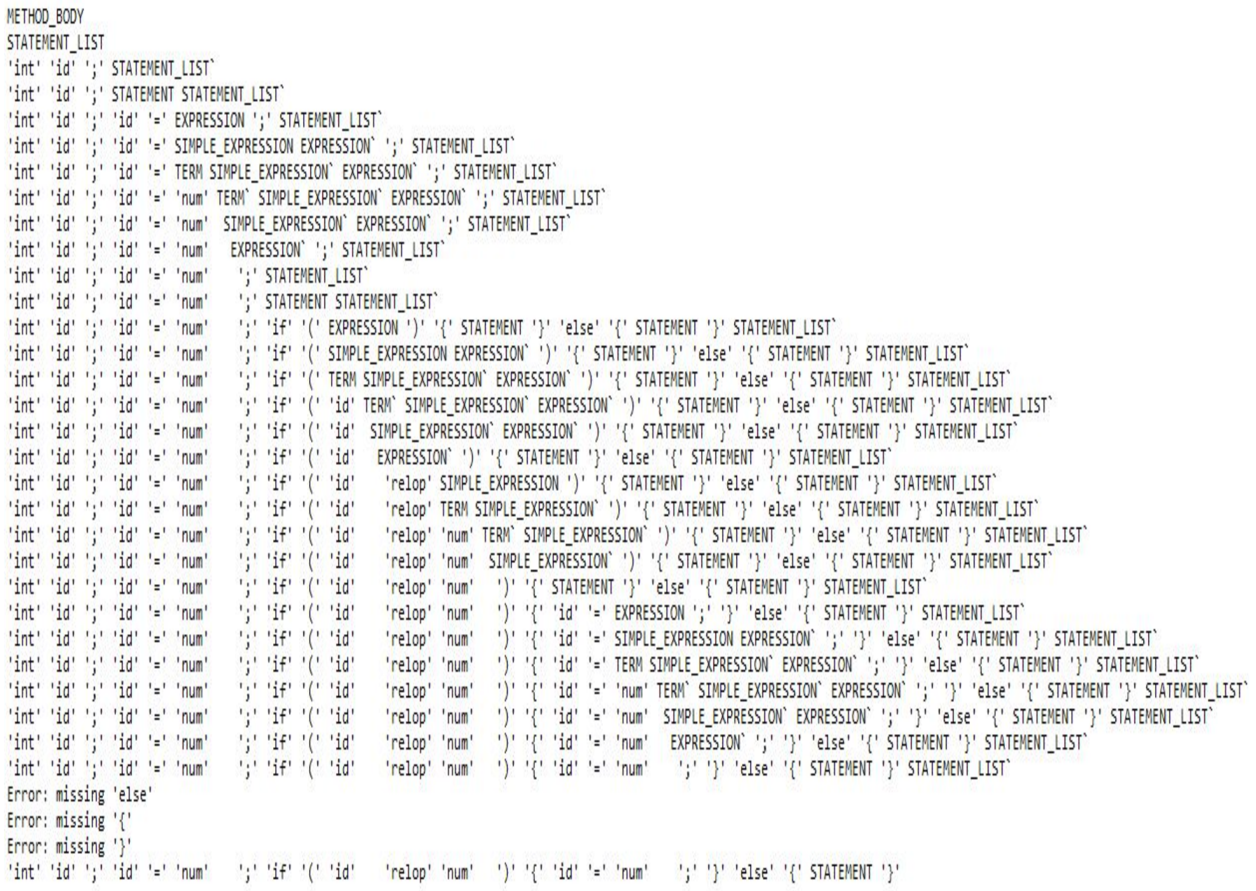
```
int x;
x = 5;
if (x > 2)
{
 x = 0;
}
```

⇒ **The output of the lexical:**



```
int
id
;
id
assign
num
;
if
(
id
relop
num
)
{
id
assign
num
;
}
```

⇒ The output of the Parser:

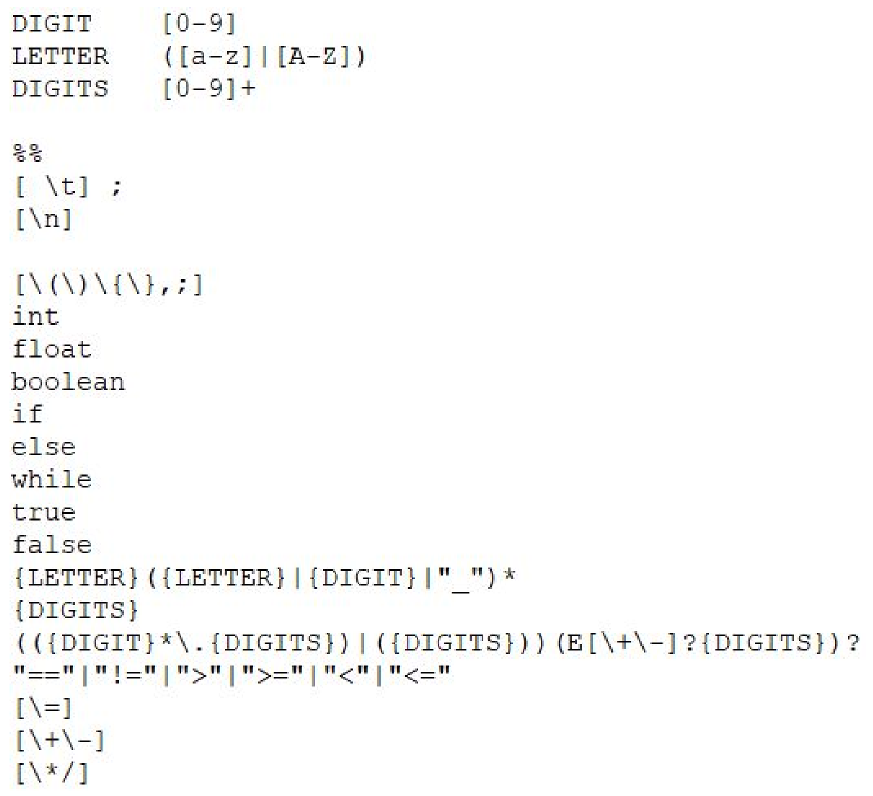


```
METHOD_BODY
STATEMENT_LIST
'int' 'id' ';' STATEMENT_LIST`
'int' 'id' ';' STATEMENT STATEMENT_LIST`
'int' 'id' ';' 'id' '=' EXPRESSION ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' SIMPLE_EXPRESSION EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' TERM SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num' TERM` SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'  SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'   EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' STATEMENT STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' SIMPLE_EXPRESSION EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' TERM SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id' TERM` SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'  SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'   EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' SIMPLE_EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' TERM SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num' TERM` SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'  SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' EXPRESSION ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' SIMPLE_EXPRESSION EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' TERM SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num' TERM` SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'  SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'   EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'    ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
Error: missing 'else'
Error: missing '{'
Error: missing '}'
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'    ';' '}' 'else' '{' STATEMENT '}'
```

/***Link to the parsing will be added below in parsing table section***/

- **Phase Three**

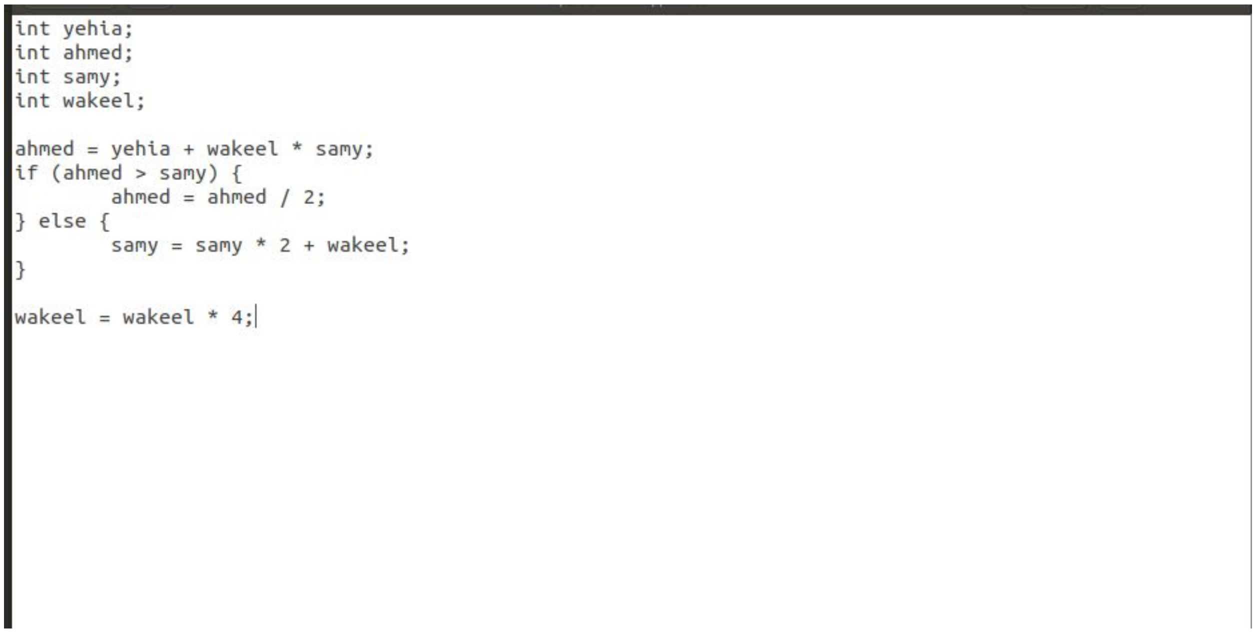
⇒ **Lexical Rules:**



```
DIGIT     [0-9]
LETTER    ([a-z]|[A-Z])
DIGITS    [0-9]+

%%
[ \t] ;
[\n]

[\(\)\{\},;]
int
float
boolean
if
else
while
true
false
{LETTER}({LETTER}|{DIGIT}|"_")*
{DIGITS}
(({DIGIT}*\.{DIGITS})|({DIGITS}))(E[\+\-]?{DIGITS})?
"=="|"!="|">"|">="|"<"|"<="
[\=]
[\+\-]
[\*/]
```

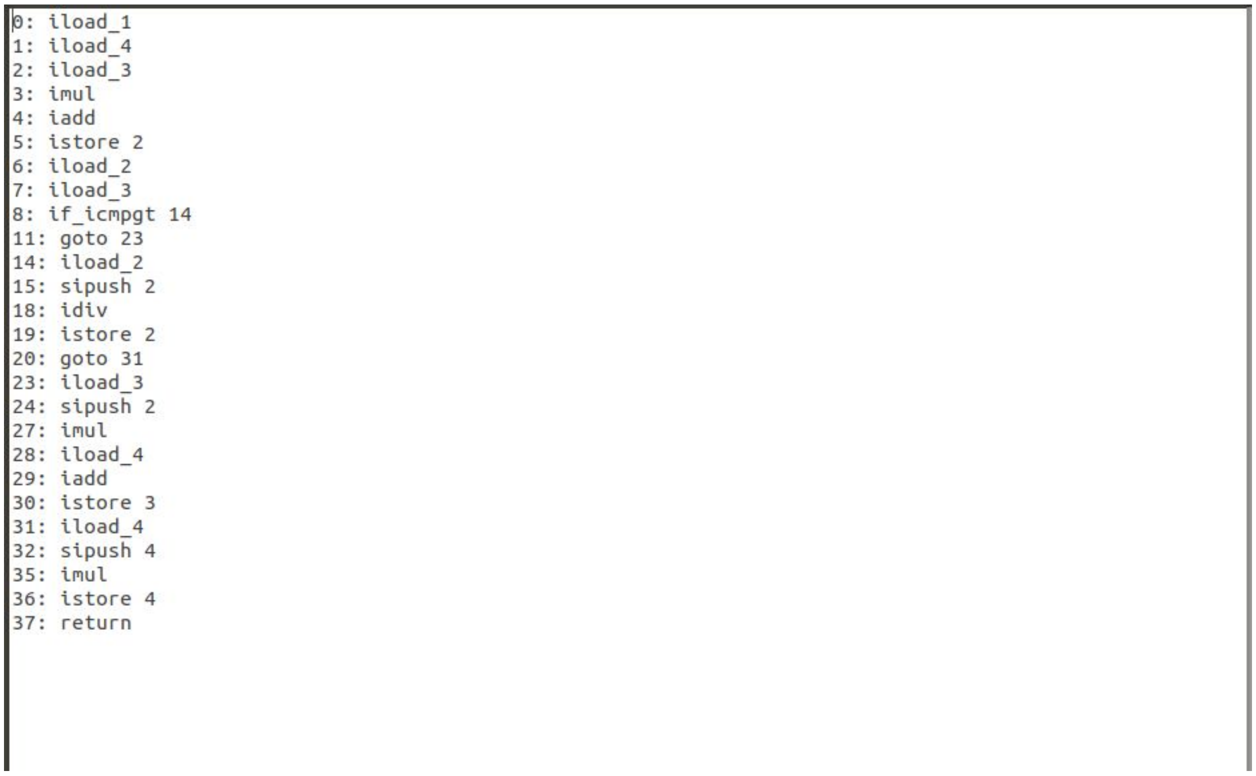
⇒ **CFG Rules: The same as Phase 2 except the part of bonus.**

⇒ Input Test Code :



```
int yehia;
int ahmed;
int samy;
int wakeel;

ahmed = yehia + wakeel * samy;
if (ahmed > samy) {
        ahmed = ahmed / 2;
} else {
        samy = samy * 2 + wakeel;
}

wakeel = wakeel * 4;
```

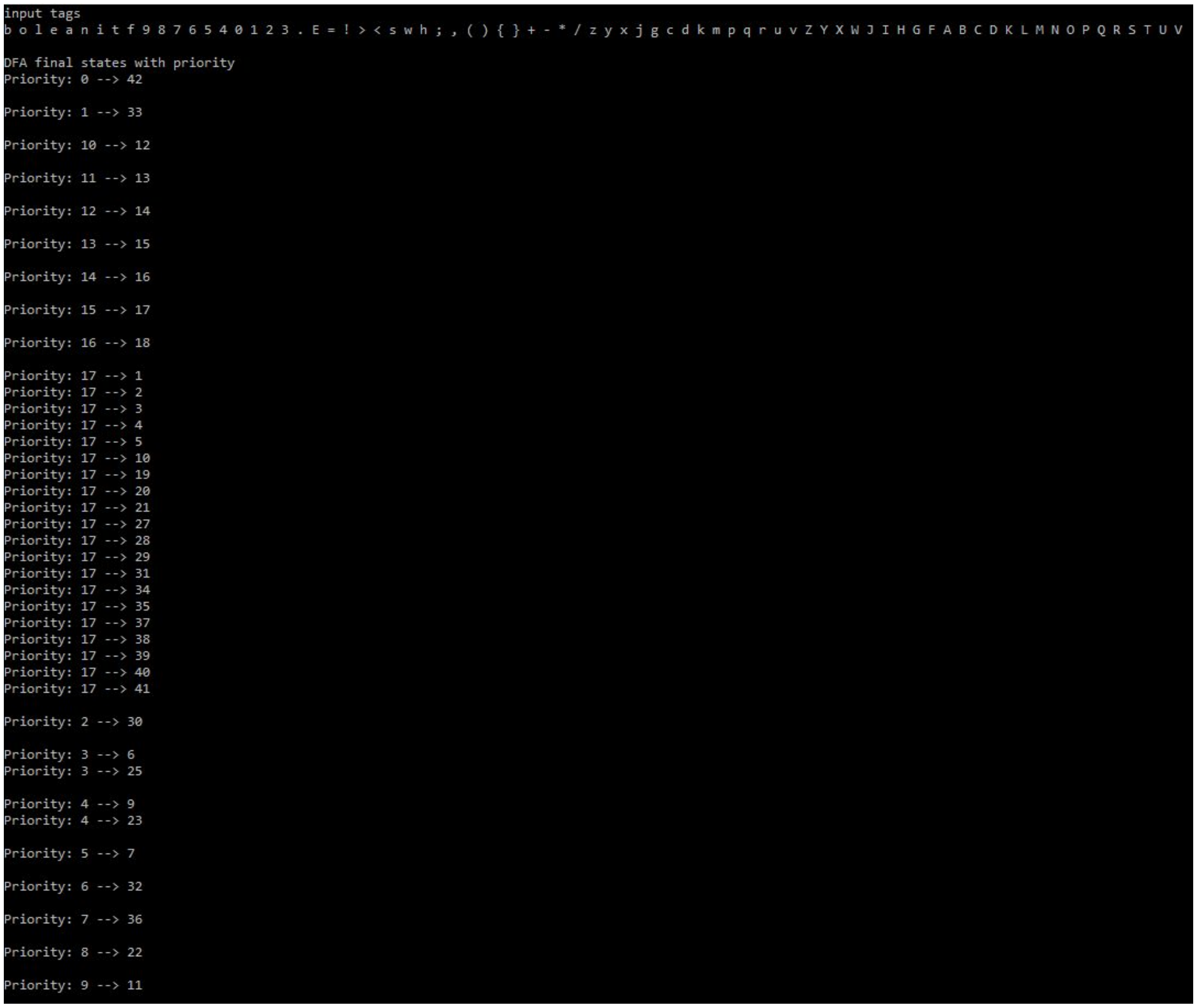
⇒ Generated Java Byte Code:



```
0: iload_1
1: iload_4
2: iload_3
3: imul
4: iadd
5: istore 2
6: iload_2
7: iload_3
8: if_icmpgt 14
11: goto 23
14: iload_2
15: sipush 2
18: idiv
19: istore 2
20: goto 31
23: iload_3
24: sipush 2
27: imul
28: iload_4
29: iadd
30: istore 3
31: iload_4
32: sipush 4
35: imul
36: istore 4
37: return
```

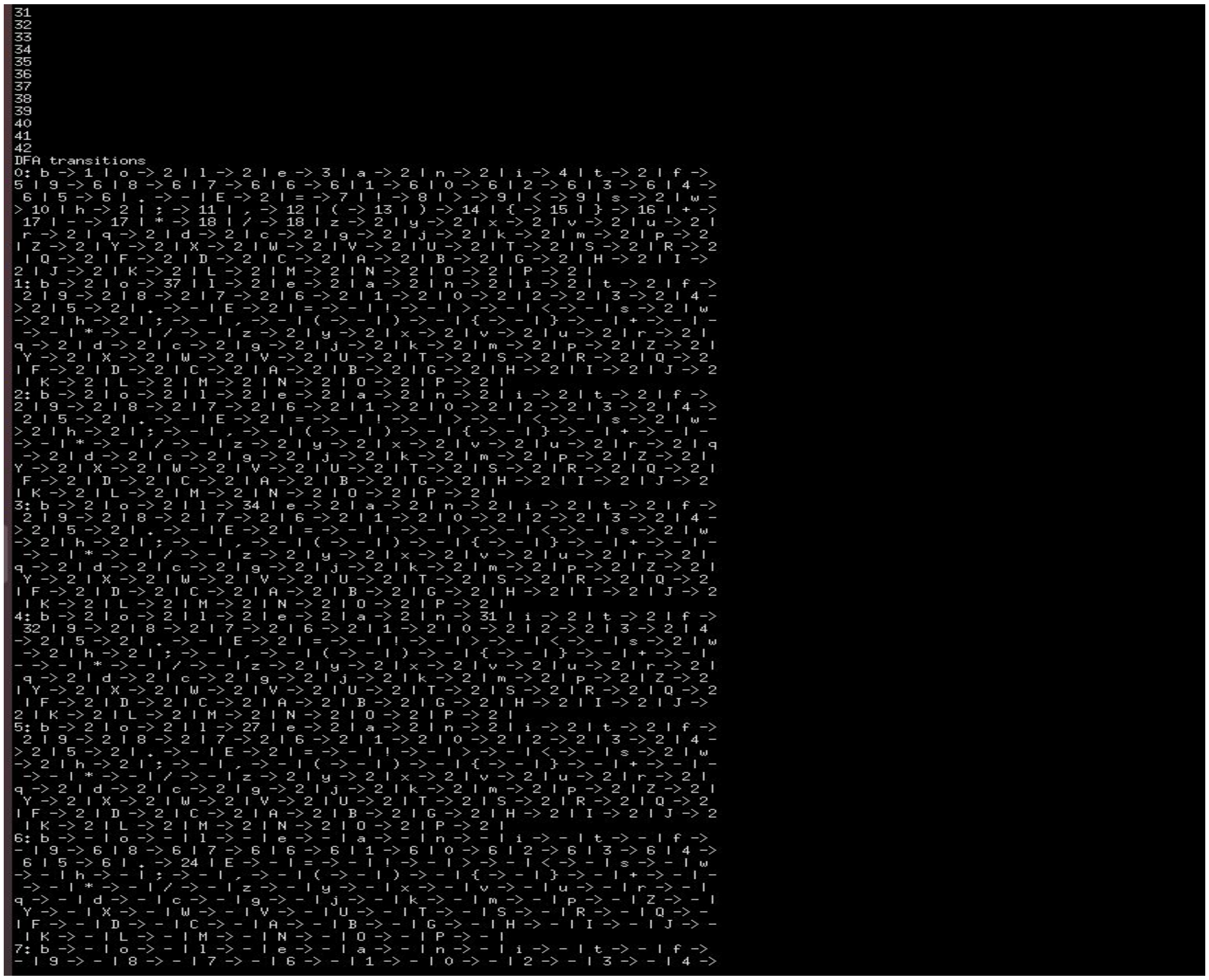
7. The resultant transition table for the minimal DFA in part 1

- **Resultant of the minimal DFA with priorities**



```
input tags
b o l e a n i t f 9 8 7 6 5 4 0 1 2 3 . E = ! > < s w h ; , ( ) { } + - * / z y x j g c d k m p q r u v Z Y X W J I H G F A B C D K L M N O P Q R S T U V

DFA final states with priority
Priority: 0 --> 42

Priority: 1 --> 33

Priority: 10 --> 12

Priority: 11 --> 13

Priority: 12 --> 14

Priority: 13 --> 15

Priority: 14 --> 16

Priority: 15 --> 17

Priority: 16 --> 18

Priority: 17 --> 1
Priority: 17 --> 2
Priority: 17 --> 3
Priority: 17 --> 4
Priority: 17 --> 5
Priority: 17 --> 10
Priority: 17 --> 19
Priority: 17 --> 20
Priority: 17 --> 21
Priority: 17 --> 27
Priority: 17 --> 28
Priority: 17 --> 29
Priority: 17 --> 31
Priority: 17 --> 34
Priority: 17 --> 35
Priority: 17 --> 37
Priority: 17 --> 38
Priority: 17 --> 39
Priority: 17 --> 40
Priority: 17 --> 41

Priority: 2 --> 30

Priority: 3 --> 6
Priority: 3 --> 25

Priority: 4 --> 9
Priority: 4 --> 23

Priority: 5 --> 7

Priority: 6 --> 32

Priority: 7 --> 36

Priority: 8 --> 22

Priority: 9 --> 11
```

- **Result transition table for the minimal DFA**



```
31
32
33
34
35
36
37
38
39
40
41
42
DFA transitions
0: b -> 1 | o -> 2 | l -> 2 | e -> 3 | a -> 2 | n -> 2 | i -> 4 | t -> 2 | f ->
5 | 9 -> 6 | 8 -> 6 | 7 -> 6 | 6 -> 6 | 1 -> 6 | 0 -> 6 | 2 -> 6 | 3 -> 6 | 4 ->
 6 | 5 -> 6 | . -> - | E -> 2 | = -> 7 | ! -> 8 | > -> 9 | < -> 9 | s -> 2 | w -
> 10 | h -> 2 | ; -> 11 | , -> 12 | ( -> 13 | ) -> 14 | { -> 15 | } -> 16 | + ->
 17 | - -> 17 | * -> 18 | / -> 18 | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 |
 r -> 2 | q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2
| Z -> 2 | Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2
 | Q -> 2 | F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I ->
2 | J -> 2 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
1: b -> 2 | o -> 37 | l -> 2 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
 2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 -
> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
-> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
 Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
| F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
2: b -> 2 | o -> 2 | l -> 2 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 ->
 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w -
> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
-> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 | q
 -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2 |
 F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
| K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
3: b -> 2 | o -> 2 | l -> 34 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
 2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 -
> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
-> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
 Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
| F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
4: b -> 2 | o -> 2 | l -> 2 | e -> 2 | a -> 2 | n -> 31 | i -> 2 | t -> 2 | f ->
 32 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4
-> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
 -> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - |
- -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
 q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2
| Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
 | F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J ->
2 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
5: b -> 2 | o -> 2 | l -> 27 | e -> 2 | a -> 2 | n -> 2 | i -> 2 | t -> 2 | f ->
 2 | 9 -> 2 | 8 -> 2 | 7 -> 2 | 6 -> 2 | 1 -> 2 | 0 -> 2 | 2 -> 2 | 3 -> 2 | 4 -
> 2 | 5 -> 2 | . -> - | E -> 2 | = -> - | ! -> - | > -> - | < -> - | s -> 2 | w
-> 2 | h -> 2 | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> 2 | y -> 2 | x -> 2 | v -> 2 | u -> 2 | r -> 2 |
q -> 2 | d -> 2 | c -> 2 | g -> 2 | j -> 2 | k -> 2 | m -> 2 | p -> 2 | Z -> 2 |
 Y -> 2 | X -> 2 | W -> 2 | V -> 2 | U -> 2 | T -> 2 | S -> 2 | R -> 2 | Q -> 2
| F -> 2 | D -> 2 | C -> 2 | A -> 2 | B -> 2 | G -> 2 | H -> 2 | I -> 2 | J -> 2
 | K -> 2 | L -> 2 | M -> 2 | N -> 2 | O -> 2 | P -> 2 |
6: b -> - | o -> - | l -> - | e -> - | a -> - | n -> - | i -> - | t -> - | f ->
- | 9 -> 6 | 8 -> 6 | 7 -> 6 | 6 -> 6 | 1 -> 6 | 0 -> 6 | 2 -> 6 | 3 -> 6 | 4 ->
 6 | 5 -> 6 | . -> 24 | E -> - | = -> - | ! -> - | > -> - | < -> - | s -> - | w
-> - | h -> - | ; -> - | , -> - | ( -> - | ) -> - | { -> - | } -> - | + -> - | -
 -> - | * -> - | / -> - | z -> - | y -> - | x -> - | v -> - | u -> - | r -> - |
q -> - | d -> - | c -> - | g -> - | j -> - | k -> - | m -> - | p -> - | Z -> - |
 Y -> - | X -> - | W -> - | V -> - | U -> - | T -> - | S -> - | R -> - | Q -> -
| F -> - | D -> - | C -> - | A -> - | B -> - | G -> - | H -> - | I -> - | J -> -
 | K -> - | L -> - | M -> - | N -> - | O -> - | P -> - |
7: b -> - | o -> - | l -> - | e -> - | a -> - | n -> - | i -> - | t -> - | f ->
- | 9 -> - | 8 -> - | 7 -> - | 6 -> - | 1 -> - | 0 -> - | 2 -> - | 3 -> - | 4 ->
```

8. Parsing Tables in part 2

- **Predictive Table**

https://drive.google.com/open?id=1BxnKjMNiGTqJ7yYdCZh82F5AWSu1pDA

- **Parsing Process**

https://drive.google.com/open?id=1cLRT7uVqxxUUZHDvbRpGtxxs995JbsKy

** The file is too big (too many lines) to be put in the report **

9. Comments about used tools

- **Flex** : we used to generate the token for the given rules in the lexical file that we used to replace the lexical analyzer from phase one and the output of it will be used through bison.
- **Bison** : We used it to generate the byte code as it takes the tokens from that generated from Flex and semantic rules and action from the parser file then generates the byte code for them.

11. Assumptions

❖ **Phase One**

➢ Range operation valid only between English characters (both capital or small) or between digits from 0 to 9. And should the left character be less than the right character like b-d or 2-8. Any other way to describe range operation will be considered a regular character

➢ When a rule in regular expression has an error then our code will ignore this rule.

❖ **Phase Two**

➢ Since we have to split the rules by the “ = “ sign and some rules have “ = “ among its production we treat the “ = “ as “ assign “ word when building the predictive table.

❖ **Phase Three**

➢ For Floats We support the arithmetic operations of the type of float that does not have E in its representation.

12. Notes

- Classes of phase one (Source And Header) exist in folder “ lexical “.
- Classes of phase Two (Source And Header) exist in folder “parser”.
- Executable of phase one & two in (Bin ⇒ Debug ⇒ Compliress.exe).
- Files of Phase three (lexical.l and parser.y and other generated files in Phase3 folder)
- For phase one: “Input.txt” has the input program, “Test.txt” has the rules and “Output.txt”
- For Phase Two: “testCase_parser” for the test cases, “parsing process” and “predictive table” for the output.
- For running phase three from CMD :
  - flex lexical.l
  - bison -d parser.y
  - g++ -std=c++11 parser.tab.c lex.yy.c -lfl -o parser
- The output of phase three (java byte code) in Output.Class

13. Conclusion

The front end phase in the compiler design is a critical phase as it analyzes the source code to build an internal representation of the program, called the intermediate representation(In our case Java Byte Code). It also generates the symbol table.

14. The role of each student

Phase	Requirement	Team member
One	1. Parsing the input file of the regular expression & Building NFA 2. Building minimized DFA 3. Generation of tokens	1. Ahmed Nabil & Mohamed Samy 2. Abdelrahman Alsayed & Yehia Elsayed 3. Ahmed Nabil(Main Procedure) & Abdelrahman Alsayed(Helper Functions)
Two	1. Parsing the input file of CFG grammars 2. Elimination of left recursion 3. Elimination of left factoring 4. Compute ‘First’ set 5. Compute ‘Follow’ Set 6. Building a predictive table 7. Stack procedure and error handling	1. Ahmed Nabi 2. Yehia Elsayed 3. Abdelrahman Alsayed & Mohamed Samy 4. Yehia Elsayed 5. Ahmed Nabil 6. Mohamed Samy 7. Abdelrahman Alsayed
Three	1. Prepare the ‘lex’ file 2. Primitive types - Arithmetic expressions - Boolean expression for the ‘bison’ file 3. If & While statements with backpatching	All team(As it is the first time to use the tool. So, we have coded this file together) 2. Abdelrahman Alsayed & Yehia Elsayed 3. Ahmed Nabil & Mohamed Samy

15. References

1. **Dragon Book - Compilers Principles Techniques and Tools (2nd Edition)**
2. **http://www.site.uottawa.ca/~caropres/SEG2106/ch08_seg2101.ppt**
3. **Lectures From Piazza**